JOURNAL OF MECHANICS OF CONTINUA AND MATHEMATICAL SCIENCES
www.journalimcms.org

ISSN (Online) : 2454 -7190 , Special Issue, No.-6, January (2020) pp 26-40 ISSN (Print) 0973-8975

INSTRUCTION SET EXTENSION OF NIOS II FOR FLOATING -POINT HOG DESCRIPTION AND IMPLEMENTATION ON AN FPGA

Anil Seker[1], BernaOrsYalcin[2]

[1]Aselsan Inc.

[2]Istanbul Technical University

siddikaseker@aselsan.com.tr, aseker@aselsan.com.tr

https://doi.org/10.26782/jmcms.spl.6/2020.01.00003

Abstract

Human detection is one of the hot topic in the field of computer vision. HOG descriptor is a widely accepted local feature extractor with high accuracy and it has heavy computation blocks in processing. Therefore, its application takes a long processing time. To improve execution time of algorithm, one of the methods is hardware acceleration. In this paper, we propose an application-specific HOG descriptor architecture on FPGA with a soft processor called as Nios II. It has the ability of instruction set extension to its base micro-architecture without any modification on the core. We select HOG specific custom instruction sets to extend. To obtain custom instruction set, we used DAG representation which is generated by LLVM compiler. The algorithm is applied on the only-processor architecture and on the proposed architecture with instruction set extension. The total execution time is measured using hardware clock counter to approximate real time consumption. The results of both architecture are compared in terms of clock count. Obviously, proposed architecture which has fully floating-point calculation is accelerated 17.68 times in comparison with pure software implementation of HOG descriptor. The implementation of the architecture is applied for 640x480x8bit test frame on low-cost Cyclone V FPGA platform.

Keywords : Custom instruction, FPGA, ASIP, HOG, hardware accelerator

I. Introduction

Pedestrian detection issue is an important topic in the field of computer vision mainly in traffic assistance systems [I], surveillance systems [II] and autonomous robotic systems [III]. There are many algorithms in the literature which extract different features of the image. The most critical requirements of the pedestrian feature extraction is high accurate computation and robust performance [IV]. One of the common algorithms which has high accuracy and robustness was

The Paper Presented at 5th International Conference on Recent Trends in Computer Sciences and Electronics (RTCSE)
Organized by University of Hawaii, USA and Gyancity Research Lab, Haryana, India

offered by Dalal and Triggs in [V], called as Histogram of Gradient (HOG). Pure software implementation of HOG provides speed and flexibilityon the design stage of the algorithm. However, the pure software implementation of HOG is not enough to meet real-time requirements without hardware extension. Field Programmable Gate Arrays (FPGAs) are one of the best candidates for hardware acceleration in comparison with pure central processing unit (CPU)implementation.

There are many various types of design of HOG algorithm on the embedded systems. Kadota et.al propose pure FPGA implementation of HOG algorithm[VI]. In their work, the simplification of "arctangent" and "square root" operations is applied which causes accuracy degradation. Bauer et.al design FPGA-GPU(Graphical Processor Unit) combination for HOG descriptor with a Gaussian kernel support [VII]. Komorkiewicz et.al bring a floating-point approach due to detection rate consideration [VIII]. His work shows that floating-point operations can effectively be applied on FPGA fabric. Kelly et.al develop HOG front end processor on FPGA. In their design, multi-core IPPro(Image Processing Processor) can reach the performance of real-time process with the software programmability [IX]. Javier et.al offers an architecture with FPGA-CPU combination on system-on-chip structures [X]. They built architecture with an operating system on ARM processor. Jose et.al, propose that multi-core soft processor system on FPGA for implementation of HOG algorithm [XI]. The task-level pipelining system achieves a speedup of 72.4× in comparison with pure software implementation on Nios II processor.

According to our knowledge, even though a few application-specific processors have been designed for HOG algorithm, none of them has custom instruction extension to base processor micro-architecture. In this paper, we offer a HOG algorithm specific-processor design with custom instruction extension to the base architecture of the processor. Moreover, we have used single-precision floating-point calculation in order to reach high accuracy without any optimization on the MATLAB-like algorithm. Firstly, C code of the algorithm is implemented on Nios II processor as pure software.

Having analyzed of pure software in terms of clock cycle, FPU (floating-point unit) and custom instruction extension which mainly consist of floating-point arithmetic are added to base processor architecture. To decide custom instruction set of algorithm, we have used open-source compiler tool low-level virtual machine (LLVM). A data-acyclic-graph (DAG) approach is used which is the output of LLVM compiler DAG generator. We have manually selected identical branches from DAG, then we have implemented them on the hardware part.The results of the pure software implementation and the hardware extension implementations are compared in terms of clock cycle.

This paper is organized as follows; an overview of the HOG algorithm is described in section 2. The implementation of custom instruction is presented in section 3. And, finally results are showed in section 4.

II. Histogram of Gradient Overview

The main idea behind the algorithm is to calculate the oriented gradient of localized areas in the input image [V]. After computing gradients, they would generate an oriented histogram. In the end, oriented histogram is normalized in

order to suppress the effect of changes on the contrast and brightness in the image.

HOG operates in a detection window, Figure 1 shows the structure of the detection window which consists of the cells and the blocks. In this study, the cell contains 8×8 pixels and the block contains 2×2 cells. The proposed detectionwindow is selected as 64×128 pixels as proposed in Dalalet. al. original paper [5]. The blocks slide rightwards and downwards (from top left to right down) direction in the input image.

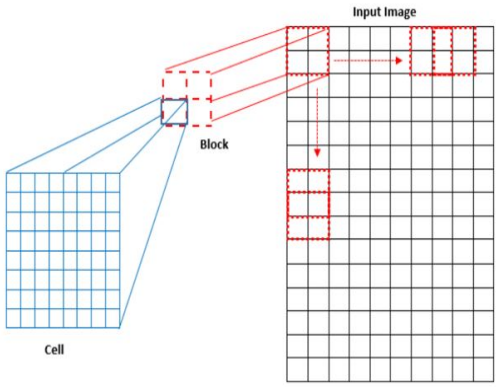


Figure. 1. HOG detection window

HOG descriptor consists of the following three main steps.

II.i. Gradient Computation

In a cell, the gradient of horizontal and vertical pixels is calculated. The main point of gradient calculation in the detection window is consists of the calculation of neighbor pixels difference. In Figure 2, the target pixel for gradient calculation is represented as blue, “x” shows the column number of pixel and “y” shows the row

number of pixels. The difference between horizontal and vertical pixels yields the changes of pixel values.

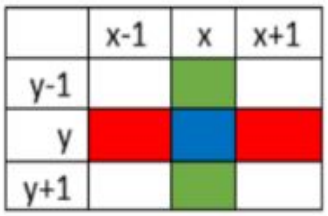


Figure. 2. Neighbor pixels

Mathematical representation of Figure 2, where $g(x,y)$ is the value of pixel at (x,y)position;

$$f_x(x,y) = g(x+1,y) - g(x-1,y) \quad (1)$$

$$f_y(x,y) = g(x,y+1) - g(x,y-1) \quad (2)$$

$f_x(x,y)$ and $f_y(x,y)$ means that color difference between neighbor pixels. The gradient is a vector so it has magnitude and direction. The magnitude equation of gradient is;

$$M\left(x,y\right) = \sqrt{f_x^2(x,y) + f_y^2(x,y)} \quad (3)$$

Direction is;

$$\theta(x,y) = \tan^{-1}\frac{f_x(x,y)}{fy(x,y)} \quad (4)$$

As it is seen from mathematical expressions, it has heavy computation processes in order to calculation gradient. The meaning of "heavy computation" in hardware is that flow consumes so many instructions,in other word clock cycles, during operation of CPU.

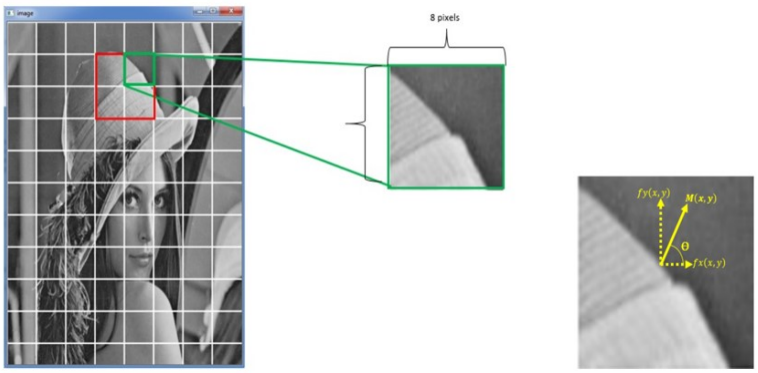


Figure. 3. Magnitude representation on the image

2.2. Histogram Generation

Histogram generation is operated as cell by cell approach. For 8×8 cell size, there will be 64 magnitude values and direction values in the histogram array. The main purpose of histogram generation is to assign gradient values to bins in the histogram map according to their directions.

The orientation bins spread over 0^o to 180^o by mirroring the gradients from 180^o–$360^o$to 0^o–$180^o$to reduce aliasing, both two neighboring bins around the direction accumulated [XII]. For 9 bins, each bin span over 20 degrees. If the degree of accumulated gradient90^owhich is exactlybetween $80^o$and 100^o, magnitude is shared at same weight to $80^o$and 100^o.

In Figure 4, there are 8 values in a row and 8 values in a column. These values represent the gradients and the directions of each pixel in the target cell. The value of magnitude is voted according to the corresponding value of direction in the matrix. If the corresponding direction does not exactly match with the divided angles between $0^o$and 180^o, the magnitude value is shared by weights. As it is seen in Figure 4, the angle value of magnitude at the position (4, 7) is 40^o. It matches the value at the binning map. Therefore, the magnitude value is directly assigned to the corresponding angle in the orientation map. The angle value of magnitude at the position (8, 1) is 130^o. It is exactly between 120^o and 140^o so the magnitude value is shared by half to 120^o and 140^o in the orientation map. Another angle value of magnitude at the position (4, 8) is 180^o. This value is assigned to 0^o due to mirroring. After binning of gradient to histogram, the local histogram which has 64 magnitude in a cell is accumulated to 9 bin values. The local histogram contains a block values that has four cells. Each block

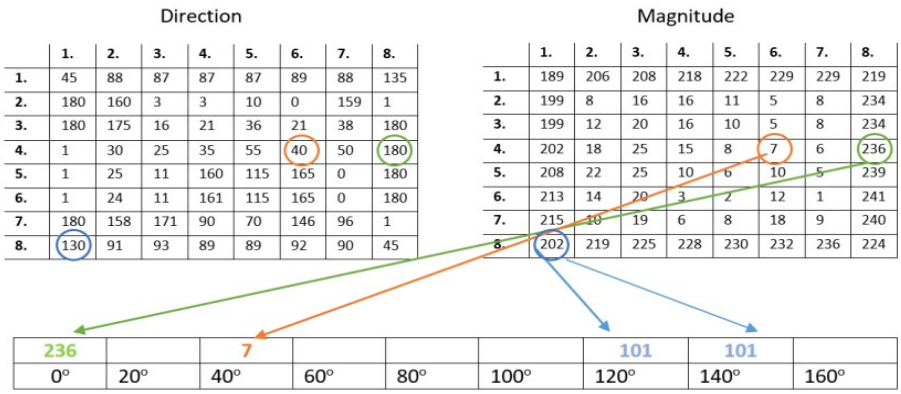


Figure. 4. Binning orientation of gradients

II.iii. Normalization

Dalal and Triggs says "Gradient strengths vary over a wide range owing to localvariations in illumination and foreground-background contrast, so effective localcontrast normalization turns out to be essential for good performance". According tothem, L2 normalization of one block which is composed of 2×2 cells yields a good performance.

Hence, L2 normalization technique will be used in this thesis. The histogram ofgradients in a cell is concatenated with other cells in a block. Each block is consists of4×9 = 36 magnitude values.

V_k: The vector elements of concatenated of four cells

$$v = \frac{V_k}{\sqrt{V_k^2+\epsilon}} \tag{5}$$

As shown in Figure 5, block1 and block2 are overlapped. As a result of normalization, changes on the pixel difference values do not affect histogram ofgradient characterization. In other word, the result of feature extraction gets morerobust. In Figure 5, all cells are represented as Ci. For block1, all cells are concatenated in one vector as $[C1C2C3C4]$ and block 2 are concatenated in one vector as $[C2C3C5C6]$.

C1 C2 C5

C4 C3 C6

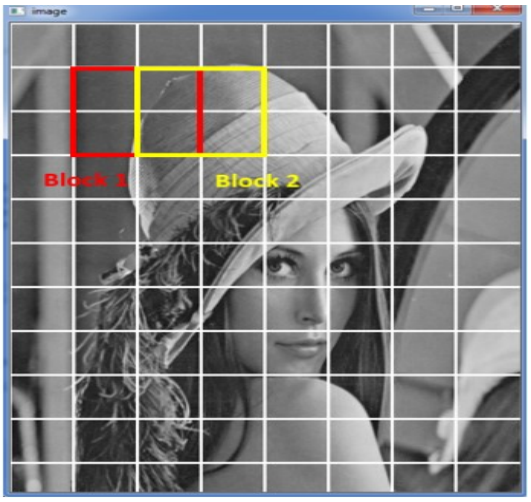


Figure. 5. Blocks representation on image

III. Instruction Set Extension of Nios II for HOG Algorithm

In this chapter, we have described the architecture of hardware/software co-design with extended instructions to Nios II base processor. The steps of implementation are;

1- Reference C design is compiled on the desktop PC with an example frame than its results are texted for testing of subsequent design on the embedded system.
2- After that, the reference design is modified for embedded system.
3- Having an analysis of proposed software architecture in terms of clock cycle, instruction flow of algorithm is profiled using open-source LLVM compiler.
4- Finally, the profiled instruction subset of algorithm is implemented on FPGA using VHDL (Very High Speed Integrated Circuit Hardware Description Language).

As a result of instruction flow analysis, the appropriate subset of instruction set is implemented on the hardware. For this implementation, we used Cyclone V GT development kit provided by Altera [XIII]. The hardware architecture is implemented by using VHDL.

III.i. Software Architecture

As explained before, HOG algorithm is consist of three main functional units respectively a) gradient calculation, b) histogram generation and c) block normalization. The parameters of algorithm are applied as the original paper of Dalalet. al. on the image of 640×480 [V]

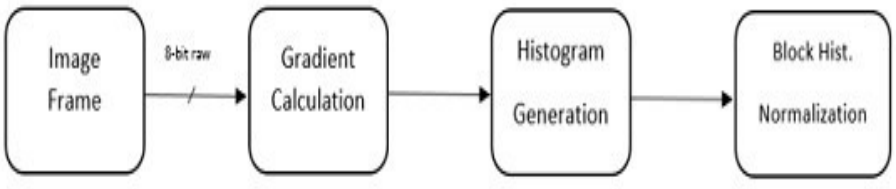


Figure. 6. HOG descriptor flow

The test frame is stored in synthesized ROM (Read-Only Memory) of FPGA in the size of 640×480×8 bit. The Nios II processor core gets access to memory in the code and reads pixel values. In orientation process, trigonometric functions are used to create orientation bins. In this design, undirected orientation angles are preferred which are between 0^o and 180^o. According to our measurement of trigonometric calculation operations take 110348 clock cycles on pure software in terms of calculating 9 bin angles. To reduce this amount, pre-calculated orientation angles memory is created to hold 9 angles values in the data type of float which is connected to Nios II processor over Avalon MM interface [XIV]. Even though memory read brings extra latency, it is still faster than direct software computation. While direct computation of trigonometric operations take 110348 clock cycles, the read of pre-calculated trigonometric values takes 122 clock cycles. We have only optimized pre-calculated binning angles as storing in memory which reduces operation time. The rest of calculations in algorithm flow are applied as single-precision floating point format.

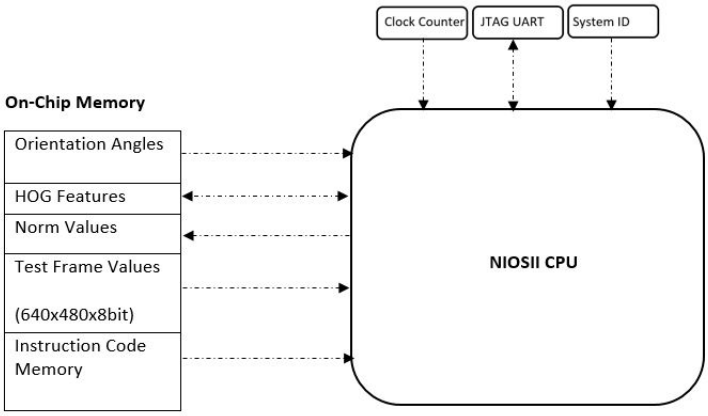


Figure. 7. Software architecture

In Figure 7, the architecture of pure software implementation is presented. The processor reads a test frame from memory and applied the algorithm.

Clock counter on the hardware part is used to measure execution time of pure software algorithm. The processor asserts a start signal over I/O connect, after finishing operation de-assert to stop counter. JTAG UART is added for debugging purposes to check results. The system is fed by 50 MHz external oscillator.

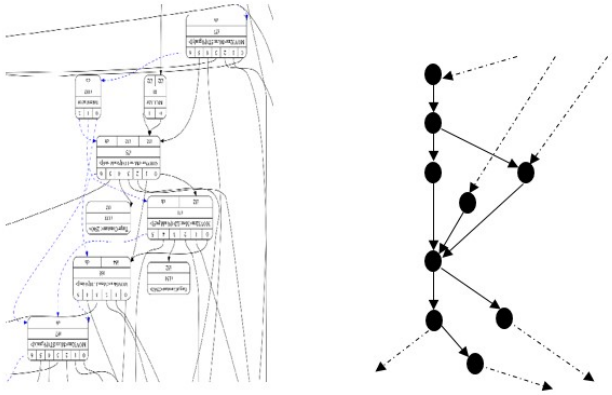
III.ii Instruction Subset Selection

In configurable processors like NiosII allows that application-specific custom instruction can be added to its Arithmetic Logic Unit (ALU) part in order to accelerate complex C/C++ applications in low design cost with software flexibility [XV]. One of the important questions is how to decide instructions for hardware extension. There are a few methods to decide instruction subset such as using commercial tools named as XPRES [XVI] and CORXpert[XVII]. In this paper, our approach is to manually select instruction set from Directed Acyclic Graph (DAG). The LLVM compiler provides target-dependent intermediate representation (IR) into object code which visually represents data flow and dependencies in DAG [XVIII].

Despite LLVM ecosystem includes many architectures like X86, X86-64, PowerPC, PowerPC-64, ARM, etc. , it does not have Nios II architecture [XIX]. Due to lack of support for Nios II architecture in LLVM tool, we have used RISC based open source processor architecture as a representative of Nios IIon LLVM compiler which is offered by Chung-Shu[20]. CPU0 is written by Verilog hardware description language for educational purposes. Due to similarities of instruction set architecture

of Nios II and Cpu0 instruction set architecture such as 32-bit architecture, having general purpose register, instruction execution stages etc., we have defined Cpu0 to LLVM compiler in order to get DAG representation of basic blocks.

The pure C algorithm of HOG is compiled with LLVM tool for CPU0. The tool divides main function to basic blocks (BBs) each is represented by a DAG. From the first instruction to the last instruction, all relations between basic blocks are shown in the DAG.



(a) **(b)**

Figure. 8. A part of original BB in DAG and its simplified representation

The LLVM tool generated fifty five BBs for HOG in total. A small part of a BB in DAG is shown in Figure 8. The black arrows imply data-dependency between instructions and black circles imply the instructions. Dashed black arrows imply data-dependency of instructions which are not represented in related DAG.

We have analyzed the repeating patterns of instructions in a DAG for adding new custom instructions to the ALU of Nios II. In order to simplify the analysis, we have transformed the original DAG output of LLVM tool to a new graph. An example graph for the DAG is shown in Figure 8(a) and transformed graph is shown in Figure 8(b).

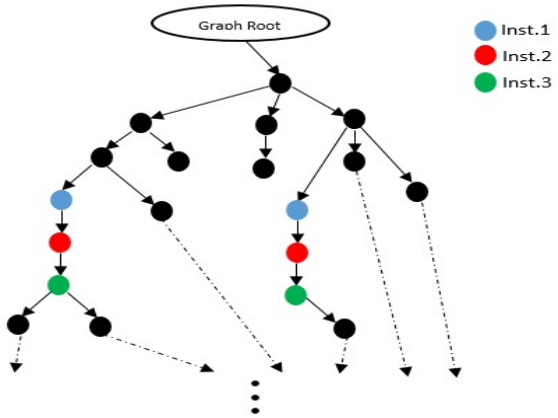


Figure. 9. Identical sub-instruction set of HOG in BB

We have produced an example transformed DAG in order to show custom instruction set detection using identical parts on branches of a DAG shown in Figure 9. The blue, red and green cycles stand for sequential instructions in identical branches of a BB which are utilized for custom instruction design. Overall, we have detected three different BBs which have identical instruction subset among fifty five BBs. These instruction subset are added in parallel to ALU that communicate with processor over custom instruction interface.

III.iii. Custom Instruction Extension Implementation

In the previous part, how custom instruction set is decided according to the identical instruction flows is explained. In total, three custom logic blocks have been designed on the hardware part in order to extend instructions of Nios II processor core. The logic blocks for implementation of custom instructions are described using VHDL and vendor-specific IP's are used for floating point arithmetic [21].

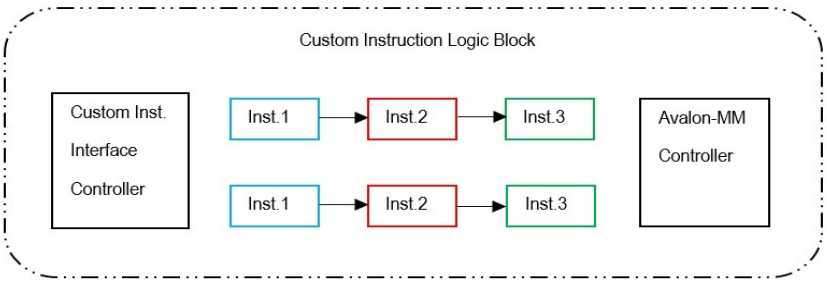


Figure. 10. Custom instruction logic block

We have designedcustom instruction logic block of instruction subset which is shown in Figure 10. Identical instruction flows are designed in parallel. Custom instruction controller is the connection between ALU and instruction subset. It is responsible for data transferring and driving control signals. We have also designed theAvalon-MM controller which is responsible for Avalon-MM interface [14]. It is a connection between custom logic block and external memory which stores operands and results. If the number of input operands is greater than two which is maximum input operand number of ALU, the processor writes operands to custom instruction memory else it transmits operands over custom instruction interface. Then, custom instruction logic block reads operands from custom instruction memory and apply instruction shown in Figure 11. After all, the processor receives result over custom instruction interface.

In Figure 11, the embedded system architecture with extended custom logic is shown. Also, there is an on-chip memory which stores pre-calculated angles, HOG features, normalization values and test frame pixel values.

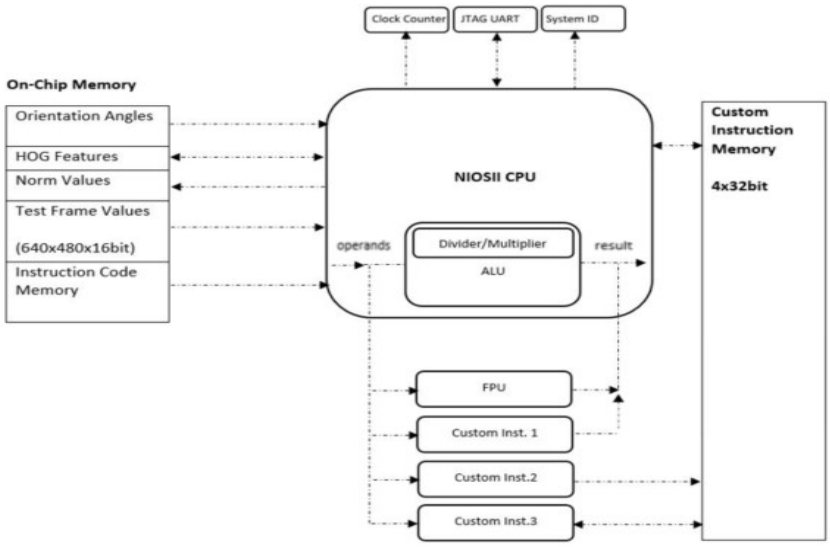


Figure. 11. Architecture with custom instruction and FPU

The three different custom instruction logic blocks which consist of instruction subset and FPU are added to ALU. Due to limit of input/output operands over interface,we have used shared 4×32-bit memory between CPU and hardware part. Custom instruction logic block 1 is directly connected to ALU. Custom instruction logic block 2 and custom instruction logic block 3 are connected between input of ALU and custom instruction memory. The extended hardware of HOG

implementation is analyzed in terms of clock cycles.The results showthat with the increasing of sources, operation time is reduced by 17.68 times in comparison with pure software implementation. Also, instruction per second is enhanced by 15.39 times according to pure software architecture.

Table 1. The Results of Architectures

Parameters	Pure Software	C.I + FPU
Clock Cycles	5.629.844.132	318.290.802
Program Size	77 Kbytes	67 Kbytes
Block Memory Bits	1.402.176	1.520.584
Adaptive Logic Modules	2.144	10.114
DSP Blocks	0	51
Fmax	180.83MHz	154.68MHz
Instruction per Second	175.06	2694.39

IV. Conclusion

We have presented a custom instruction set extension of Nios II for HOG application in this work. The shortest execution time for an algorithm can be achieved by pure hardware implementation, with the price of non-flexibility. On the other end, we can have the most flexible implementation using only software which will be the slowest.

We would like to use the advantage of both way of implementation by using application-specific processor design approach. For this purpose, we have added three custom instructions made of some instruction flows. The appropriate instruction flows have been determined by the analysis of LLVM DAG output.

The results show that the operation time of the HOG algorithm implemented on the instruction set extended Nios II is 17.68 times lower than the pure software implementation of the same algorithm. As a result of the work, it is observed that any algorithm can be enhanced by proposed method as application-specific processor if the processor core allows modification on its architecture.

References

I. A. Shashua, Y. Gdalyahu and G. Hayun, Pedestrian detection for driving assistance systems: single-frame classification and system level performance, IEEE Intelligent Vehicles Symposium, (2004)June 14-17 ;Parma, Italy.

II. Paul, Manoranjan and Haque, Shah and Chakraborty, Subrata, Human detection in surveillance videos and its applications - A review, EURASIP Journal on Advances in Signal Processing, 25, (2013) .

III. Joshi, Rajeev, Pratap Chandra Poudel and Pankaj Bhandari, An Embedded Autonomous Robotic System for Alive Human Body Detection and Rescue Operation, International Journal of Scientific and Research Publications, 4, 5,(2014).

IV. H. Ninomiya, H. Ohki, K. Gyohten and N. Sueda, An evaluation on robustness and brittleness of HOG features of human detection, Korea-Japan Joint Workshop on Frontiers of Computer Vision(FCV), (2011) February 9-11, Ulsan, South Korea.

V. N. Dalal and B. Triggs, Histograms of Oriented Gradients for Human Detection, IEEE Computer Society Conference on Computer Vision and Pattern Recognition(CVPR), (2005) June 20-25, San Diego, CA, USA.

VI. R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto and Y. Nakamura, Hardware Architecture for HOG Feature Extraction, International Conference on Intelligent Information Hiding and Multimedia Signal Processing, (2009) September 12-14, Kyoto, Japan.

VII. S. Bauer, S. Kohler, K. Doll and U. Brunsmann, FPGA-GPU architecture ¨ for kernel SVM pedestrian detection, IEEE Computer Society Conference on Computer Vision and Pattern Recognition, (2010) June 13-18San Francisco, CA, USA.

VIII. M. Komorkiewicz, M. Kluczewski and M. Gorgon, Floating point HOG implementation for real-time multiple object detection, International Conference on Field Programmable Logic and Applications (FPL), (2012) August 29-31, Oslo, Norway.

IX. Kelly, Colm and Siddiqui, Fahad and Bardak, Burak and Woods, Roger, Histogram of Oriented Gradients front end processing: an FPGA Based Processor Approach, IEEE Workshop on Signal Processing Systems (SiPS), (2014) October 20-22, Belfast, UK.

X. J. Cerezuela-Mora, E. Calvo-Gallego and S. Sanchez-Solano, Hardware/software co-design of video processing applications on a reconfigurable platform, IEEE International Conference on Industrial Technology (ICIT), (2015) March 17-19, Seville, Spain.

XI. J. A. M. de Holanda, J. M. P. Cardoso and E. Marques, A pipelined multi-softcoreapproach for the HOG algorithm, Conference on Design and Architectures for Signal and Image Processing (DASIP), (2016) October 12-14, Rennes, France.

XII. J. Wang et al., Simplifying HOG arithmetic for speedy hardware realization, IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), (2014) November 17-20,Ishigaki, Japan.

XIII. Altera (2017). Cyclone V GT FPGA Development Board Reference Manual

XIV. Intel (2018). Avalon R Interface Specifications, September 26

XV. R. Lysecky and F. Vahid, A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning, Design, Automation and Test in Europe, (2005) March 7-11 Munich, Germany.

XVI. The XPres Compiler, Retrieved from: http://www.tensilica.com, Last accessed on: 13th November 2019

XVII. The CORXpert, Retrieved from: http://www.coware.com,Last accessed on: 13th November 2019

XVIII. The LLVM Compiler Infrastructure, Retrieved From: http://llvm.org, Last accessed on: 13th November 2019

XIX. The LLVM Compiler Architectures, Retrieved From:https://llvm.org/docs/CompilerWriterInfo.html, Last accessed on: 13th November 2019

XX. C. Chung-Shu, Tutorial: Creating an LLVM Backend for the Cpu0 Architecture, http://jonathan2251.github.io/web/index.html, 2018

XXI. Intel (2018).Floating-Point Megafunctions User Guide, November 13